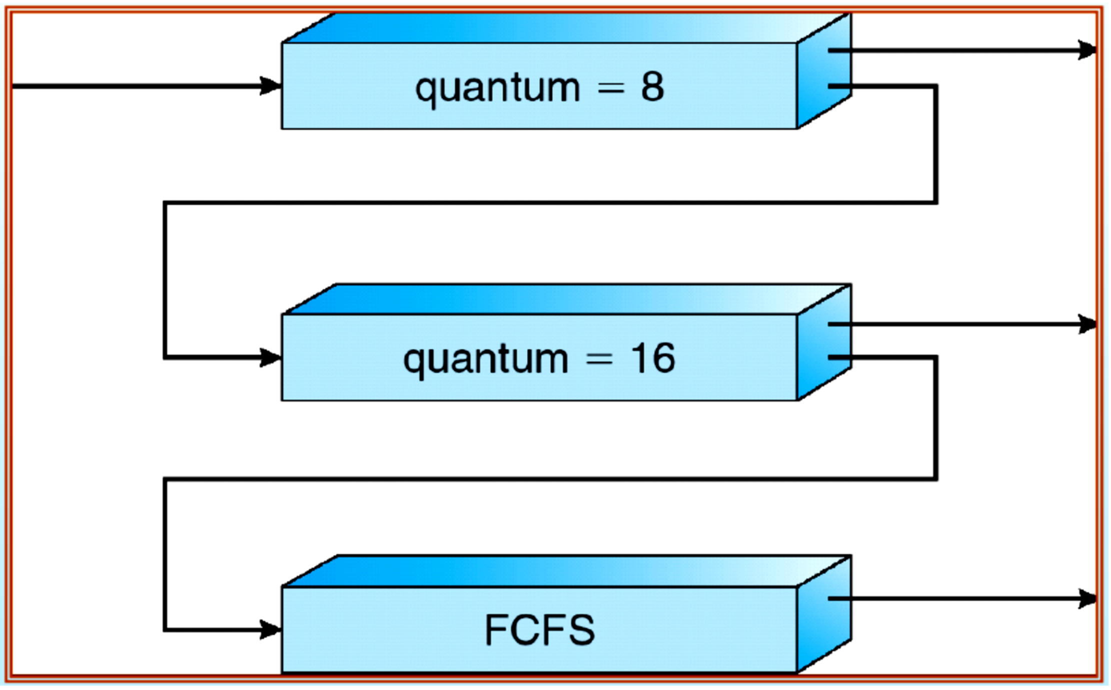
Chapter 7
CPU scheduling

This chapter will discuss the following concepts:



7.1 Basic Concepts

CPU scheduling is the basis of multi-programmed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

7.1.1 CPU-I/O Burst Cycle

Process execution consists of a cycle of **CPU** execution and **I/O** wait. Processes alternate between these two states. Process execution begins with a **CPU burst**.

That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution (Figure 7.1).

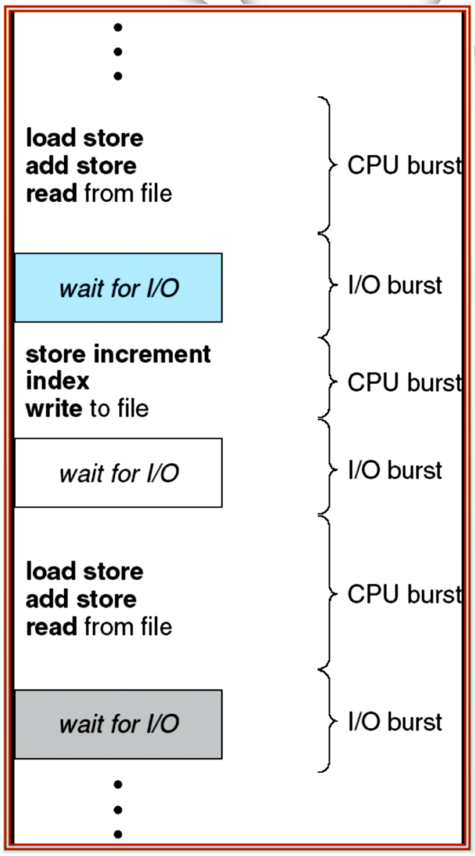


Figure 7.1 Alternating sequence of CPU and I/O bursts

7.1.2 Preemptive and non-Preemptive Scheduling

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

7.1.3 Dispatcher

Another component involved in the CPU-scheduling function is **the dispatcher**. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. The dispatcher should be as fast as possible.

7.2 Scheduling Criteria

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. Many criteria have been suggested for comparing CPU scheduling algorithms. The criteria include the following:

- **CPU utilization:** CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput:** One measure of work is the number of processes that are completed per time unit, called throughput.
- **Turnaround time:** The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

- **Waiting time:** The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends **waiting in the ready queue**. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time:** is the time from the submission of a request until the first response is **produced**. It is the time it takes to start responding, not the time it takes to **output** the response.

7.3 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms. In this section, we describe several of them.

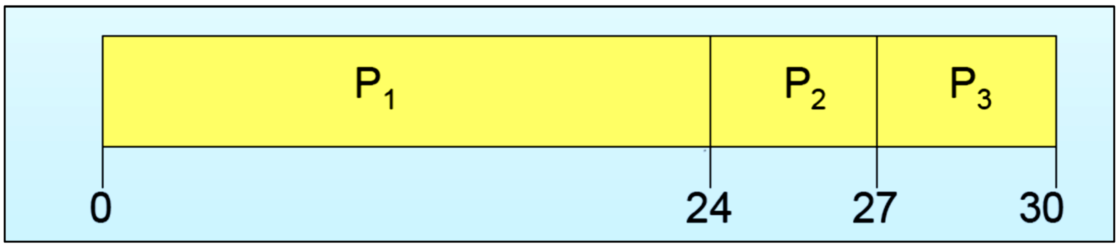
7.3.1 First-Come, First-Served Scheduling

By far the simplest CPU-scheduling algorithm is the **first-come, first-served** (**FCFS**) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue.. The average waiting time under the FCFS policy, however, is often quite long.

Example: Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following **Gantt chart**:



The waiting time is

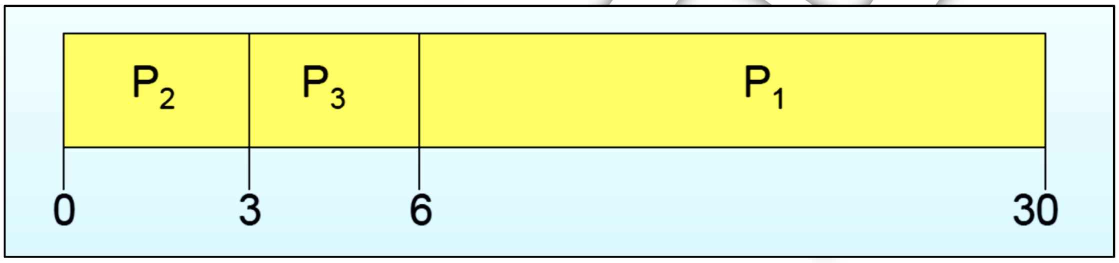
0 milliseconds for process P1,

24 milliseconds for process P2, and

27 milliseconds for process P3.

Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.

If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.

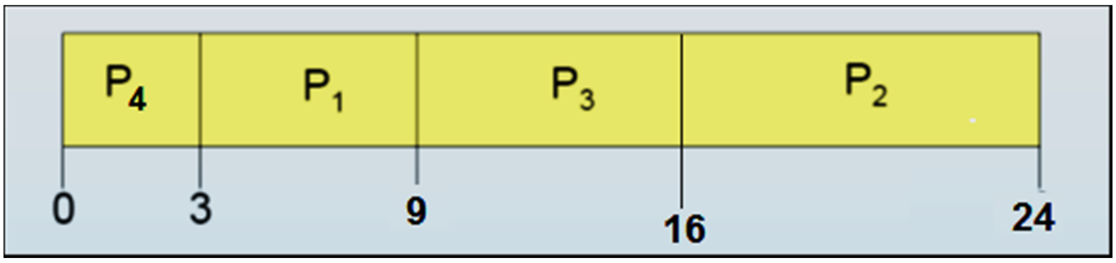
7.3.2 Shortest-Job-First Scheduling

A different approach to CPU scheduling is the **shortest-job-first** (**SJF**) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the ***smallest*** next CPU burst. If the next CPU bursts of two processes are the same, **FCFS** scheduling is used to break the tie.

Example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

process	Burst time
P_1	6
P_2	8
P_3	7
P_4	3

Using **SJF** scheduling, we would schedule these processes according to the following Gantt chart:



The waiting time is
3 milliseconds for process P1,
16 milliseconds for process P2,
9 milliseconds for process P3, and
0 milliseconds for process P4.

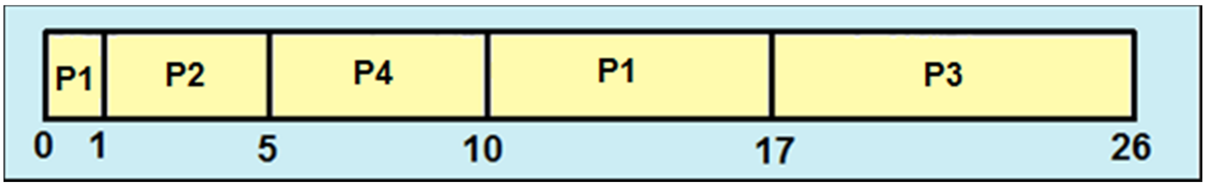
Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds. (Home work)

The SJF algorithm can be either **preemptive** or **non-preemptive**. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A **preemptive SJF** algorithm will preempt the currently executing process, whereas a **non-preemptive SJF** algorithm will allow the currently running process to finish its CPU burst.

Example of preemptive SJF, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival time	Burst time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.

The average waiting time for this example is

(0-0) + (10 - 1) for p1 time allocated to CPU –last time in CPU

(1 - 1) for p2 time allocated to CPU –Arrival time

(17 - 2) for p3 time allocated to CPU –Arrival time

(5 - 3) for p4 time allocated to CPU –Arrival time

= (9+0+15+2)/4=26/4 = 6.5 milliseconds.

Non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds. (Home work)

7.3.3 Priority Scheduling

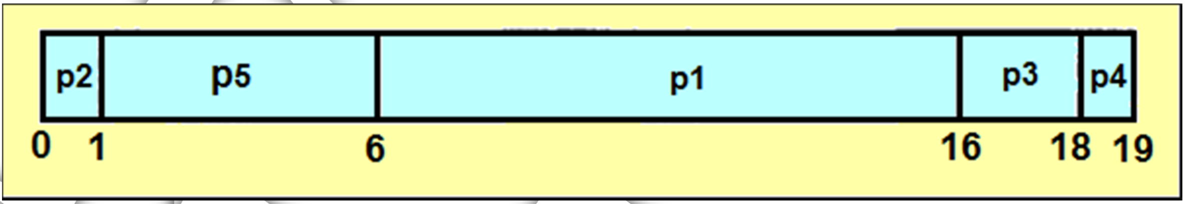
A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in **FCFS** order. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. Here, we assume that ***low numbers represent high priority.***

Example:

Consider the following set of processes, assumed to have arrived at time 0, in the order P1, P2, .. , P5, with the length of the CPU burst given in milliseconds:

Process	Burst time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds. (How?)

7.3.4 Round-Robin Scheduling

The **round-robin** (**RR**) scheduling algorithm is designed especially for time sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a **time quantum** or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

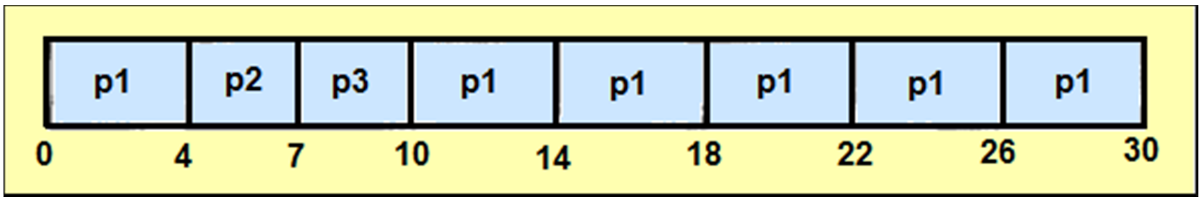
One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A **context switch** will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long.

Example: Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst time
P_1	24
P_2	3
P_3	3

If we use a time quantum of **4 milliseconds**, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2. Since process P2 does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is



The average waiting time is 17/3 = 5.66 milliseconds.

Note:

The average waiting time for p1= (0-0) + (10-4) =6 where 10 is time allocated to CPU and 4 last time of p1 for the CPU.

For p2 (4-0) = 4 (time allocated to CPU- arrival time)

For p3 (7-0) = 7 (time allocated to CPU- arrival time)

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus **preemptive**.

7.3.5 Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues (Figure 7.2). The processes are permanently assigned to one queue, generally based on some property of the process, such as **memory size**, **process priority**, or **process type**. Each queue has its own scheduling algorithm. For example, separate queues might be used by an RR algorithm, while the other queue is scheduled by an FCFS algorithm.

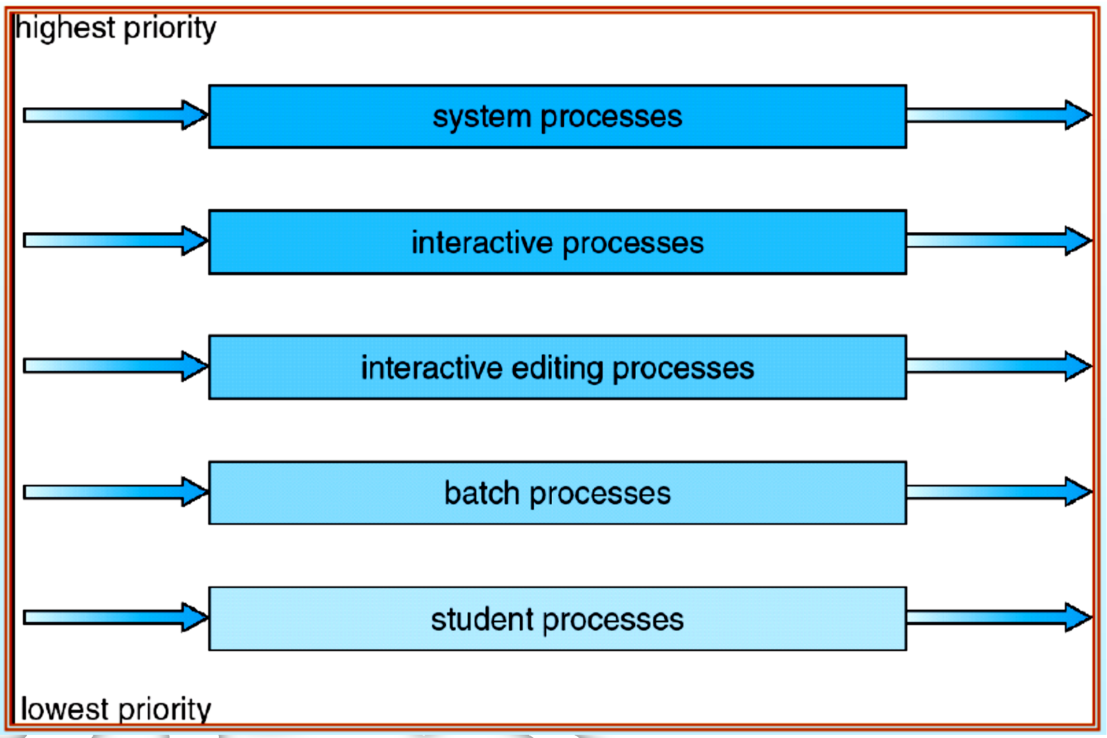


Figure 7.2 multilevel queue scheduling.

7.3.6 Multilevel Feedback-Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. Processes do not move from one queue to the other, since processes do not change their nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

The **multilevel feedback-queue scheduling algorithm**, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue.

For example, consider a multilevel feedback-queue scheduler with three queues, numbered from 0 to 2 (Figure 7.3). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

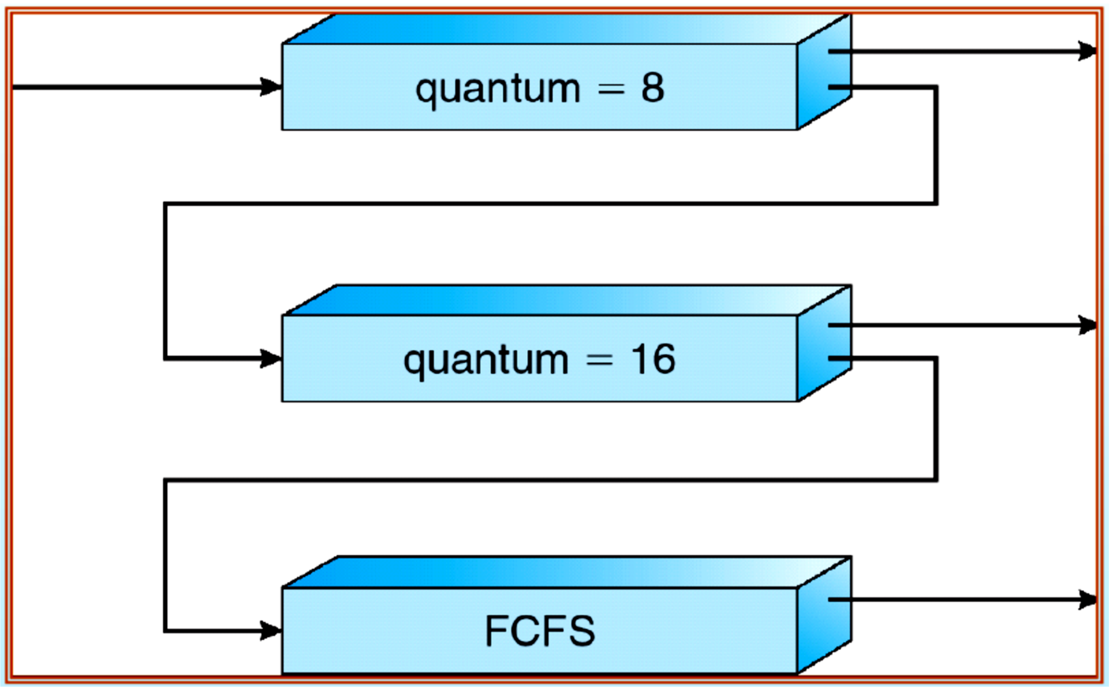


Figure 7.3 Multilevel feedback queues.

7.4 Multiple-Processor Scheduling

Our discussion thus far has focused on the problems of scheduling the CPU in a system with a single processor. If multiple CPUs are available, load sharing becomes possible; however, the scheduling problem becomes correspondingly more complex. Many possibilities have been tried; and as we saw with single processor CPU scheduling, there is no one best solution.

One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server. The other processors execute only user code. This **asymmetric** multiprocessing is simple because only one processor accesses the system data structures, reducing the need for data sharing.

A second approach uses **symmetric multiprocessing** (**SMP**), where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.

End of chapter 7